Tip 36: Try a new shell

Ben Klemens

12 December 2011

level: command-line habitant
purpose: get comfortable in your shell

I started (Entry #082) this set of shell tips with a list I made up: the shell provides (1) facilities for interactive comfort, (2) history, (3) a ton of macro-like expansions, and (4) programming standards like `for` loops and `if` statements. I then gave you tips for parts (2), (3), and (4).

Here's my tip for item (1): try a new shell. There is no particular reason for the interactive features to stick to any one standard, because it is by definition not the programmable and scriptable part of things, so some shells provide much more user comfort than others. If some shell wants to have an animated paperclip in the corner kibbitzing your work, who're they hurting?

The shell you're using now is probably bash, the Bourne-again shell, so named because it's a variant of Stephen Bourne's shell for the original UNIX. It is part of the GNU project, and so has very wide distribution. But wide distribution means that it can't be too experimental with new features and comforts, because it has to run everywhere, and for every stupid quirk somebody has written a *really important* shell script that depends upon that quirk. I have my own complaints: GNU tools tend to provide half-hearted support for the `vi` keymap (which I have very much internalized), and the manual page is amazing for having absolutely no examples.

If you are on a Mac with software a couple of years old, then you may be on a version of the C shell. The C shell is really not an acceptable shell anymore, and you will especially reap benefits from dropping it.

Notably, there's tab completion. In bash, if you type part of a file name and hit `<tab>`, the name will be autocompleted if there's only one option, and if not, hit `<tab>` again to see a list options. If you want to know how many commands you can type on the command line, hit `<tab><tab>` on a blank line and bash will give you the whole list.

There are two types of shell users: those who didn't know about this tab-completion thing, and people who use it *all the time on every single line*. Now and then I wind up on a machine (like an old Mac) that somehow doesn't have tab completion, and I can only go about fifteen minutes trying to work with it before winding up in the fœtal position.

But shells beyond bash go further. When you type `ls -<tab>` in the Z shell, it will check the help pages and tell you what command line switches are available, and when you type `make <tab>` it will read your makefile and tell you the possible targets. The Friendly Interactive shell (fish) will check the manual pages for the summary lines, so when you type `man l<tab>` it will give you a one-line summary of every command beginning with L, which may save you the trouble of actually pulling up any man page at all.

Here are a few zsh tips, that give you a hint as to what switching from bash can get you. There are lots of other shells out there; I'm writing about zsh because it's the one I know best [and it gets the vi keymap right].

You can find many pages of Z shell tips[1], or maybe check out this 14-page Zsh reference card[2] [PDF]. So there goes parsimony—but why bother being Spartan with interactive conveniences. [If you have Spartan æsthetics, then you still want to switch out of bash; try `ash`.] Much of the documentation is taken up by options you can add to your `.zshrc` (or just type onto the command line); here are the two you'll need for the examples below:

```
setopt INTERACTIVE_COMMENTS
#now commends like this won't give an error
setopt EXTENDED_GLOB
#for the paren-based globbing below.
```

Expansion of globs, like replacing `file.*` with `file.c file.o file.h` is the responsibility of the shell. The most useful way in which Zsh expands this is that `**/` tells the shell to recurse the directory tree when doing the expansion. A POSIX-standard shell reads `~` to be your home directory, so if you want every `.c` file anywhere in your purview, try

```
ls ~/**/*.c
```

Remember last time how we backed up our `.c` files? Let's do that with every last one of 'em:

```
for ff in ~/**/*.c; do cp $ff ${ff}.bkup; done

#What backups do we have now?
#you may need "ls --color=no"
ls ~/**/*.c.bkup > list_of_backups
```

The Z shell also allows post-glob modifiers, so `ls *(.)` lists only plain files and `ls *(/)` lists only directories. This gets Byzantine, so try `ls *(<tab>` to get the full list of options before typing in that last `)`.

Oh, and remember from Tip #33 (Entry #083) how bash only gives you arithmetic expansion on integers, so `$((3/2))` is always one? Zsh and Ksh (and I dunno which others) are C-like in giving you a real (more than integer) answer if you cast the numerator or denominator to float:

[1] `http://grml.org/zsh/zsh-lovers.html`

[2] `http://www.bash2zsh.com/zsh_refcard/refcard.pdf`

```
#works for zsh, syntax error for bash:
echo $((3/2))
echo $((3/2.))

 #repeating the line-count example from Tip 33:
Files=`ls *.c |wc -l`
Lines=`grep '[)};]' *.c | wc -l`
 #Cast to floating point by adding 0.0
echo lines/file = $(($Lines/($Files+0.0)))
```

Now the shell-as-desk calculator is usable again.

Spaces in file names can break things in bash, because spaces separate list elements:

```
echo t1 > "test_file_1"
echo t2 > "test file 2"

#This fails in bash, is OK in Zsh
for f in test* ; do cat $f; done
```

The Z shell has array variables (which you can define using parens) that don't rely on spaces as delimiters, so the above isn't a problem:

```
#Having made the files above, run this in zsh:
for f in test* ; do cat $f; done

#equivalent to:
files=(test*)
for f in $files ; do cat $f; done
```

OK, enough about the Z shell, which is the one that is currently working well for me. There are many more to be had, and the odds are good that there's a shell that will work better for you than the one that shipped with your box. Wikipedia[3] has a shell comparison chart. Perl fans, maybe try the Perl shell[4].

If you decide to switch, there are two ways to do it: you can use `chsh` to make the change official in the login system [`/etc/passwd` gets modified], or if that's somehow a problem, you can add `exec -l /usr/bin/zsh` (or whatever shell you like) as the last line of your `.bashrc`, so bash will replace itself with your preferred shell every time it starts.

[3]http://en.wikipedia.org/wiki/Comparison_of_computer_shells
[4]http://www.focusresearch.com/gregor/sw/psh/